

Exercise 11:

Sentiment analysis

Implement Embedding

```
class Embedding(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, padding_idx):
        """
        """
        super().__init__()
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        # We handle the padding for you
        self.padding_idx = padding_idx
        self.register_buffer(
            'padding_mask',
            (torch.arange(0, num_embeddings) != padding_idx).view(-1, 1)
        )

        self.weight = None
        #####
        # TODO: Set self.weight to a parameter initialized with standard normal #
        # N(0, 1) and has a shape of (num_embeddings, embedding_dim).          #
        #####
        self.weight = nn.Parameter(torch.randn(num_embeddings, embedding_dim))
        #####
        #                                     END OF YOUR CODE                      #
        #####

        # Handle the padding
        self.weight.data[padding_idx] = 0
```

```
def forward(self, inputs):
    """
    Inputs:
        inputs: A long tensor of indices of size (seq_len, batch_size)
    Outputs:
        embeddings: A float tensor of size (seq_len, batch_size, embedding_dim)
    """

    # Ensure <eos> always return zeros
    # and padding gradient is always 0
    weight = self.weight * self.padding_mask

    embeddings = None

    #####
    # TODO: Select the indices in the inputs from the weight tensor          #
    # hint: It is very short                                                  #
    #####
    embeddings = weight[inputs]
    #####
    #                                     END OF YOUR CODE                      #
    #####

    return embeddings
```

Here we build Embedding
for generating representations.

Implement an RNN Classifier

```
class RNNClassifier(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, hidden_size, use_lstm=True, **additional_kwargs):
        """
        """
        super().__init__()
        # Change this if you edit arguments
        self.hparams = {
            'num_embeddings': num_embeddings,
            'embedding_dim': embedding_dim,
            'hidden_size': hidden_size,
            'use_lstm': use_lstm,
            **additional_kwargs
        }
        #####
        # TODO: #####
        self.embedding = Embedding(num_embeddings, embedding_dim, 0)
        self.rnn = (LSTM if use_lstm else RNN)(embedding_dim, hidden_size)
        self.output = nn.Linear(hidden_size, 1)
        #####
        #                               END OF YOUR CODE                               #
        #####
```

- Here Embedding will be passed to the recurrent neural network.

Implement an RNN Classifier

```
class RNNClassifier(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, hidden_size, use_lstm=True, **additional_kwargs):

    def forward(self, sequence, lengths=None):
        #####
        #####
        output = None

        #####
        # TODO: Apply the forward pass of your network #
        # hint: Don't forget to use pack_padded_sequence if lengths is not None #
        # packed_padded_sequence should be applied to the embedding outputs #
        #####
        embeddings = self.embedding(sequence)
        if lengths is not None:
            embeddings = pack_padded_sequence(embeddings, lengths)
        h_seq, h = self.rnn(embeddings)
        if isinstance(h, tuple):
            h = h[0]

        output = self.output(h.squeeze(0)).sigmoid().view(-1)
        #####
        #                               END OF YOUR CODE                               #
        #####

    return output
```



- Sequences need to have the same length
- For this, we use the function *pack_padded_sequence* to truncate longer reviews and pad shorter reviews with zeros .

Implement an RNN Classifier

```
class RNNClassifier(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, hidden_size, use_lstm=True, **additional_kwargs):

    def forward(self, sequence, lengths=None):
        #####
        #####
        output = None

        #####
        # TODO: Apply the forward pass of your network #
        # hint: Don't forget to use pack_padded_sequence if lengths is not None #
        # packed_padded_sequence should be applied to the embedding outputs #
        #####
        embeddings = self.embedding(sequence)
        if lengths is not None:
            embeddings = pack_padded_sequence(embeddings, lengths)

        h_seq, h = self.rnn(embeddings)
        if isinstance(h, tuple):
            h = h[0]

        output = self.output(h.squeeze(0)).sigmoid().view(-1)
        #####
        #                               END OF YOUR CODE                               #
        #####

        return output
```



- After the RNN layer, we use sigmoid as the activation function as our task is a binary prediction.

Create model

```
from exercise_code.rnn.tests import classifier_test, parameter_test
from exercise_code.rnn.text_classifiers import RNNClassifier
model = None
#####
# TODO - Create a Model                                     #
#####
model = RNNClassifier(len(vocab), 22, 32)
#####
#                               END OF YOUR CODE           #
#####

# Check whether your model is sufficiently small and have a correct output format
parameter_test(model), classifier_test(model, len(vocab))
```

```
Total number of parameters: 117245
Your model is sufficiently small :)
All output tests are passed :)!

(True, True)
```

Remark:

Try to use other *embedding dimensions* and *hidden sizes* for better results.

Training

```
#####  
# TODO - Train Your Model #  
#####  
optim = torch.optim.Adam(model.parameters())  
epochs = 5  
gclip = 40  
# Training loop  
for e in range(epochs):  
    print('Epoch {}...\n'.format(e))  
    print('Starting training...')  
    model.train()  
    num_corrects = 0  
    num_labels = 0  
    total_loss = 0.0  
    for i, data in enumerate(train_loader):  
        seq = data['data'].to(device)  
        label = data['label'].to(device)  
        seq_lens = data['lengths']  
  
        model.zero_grad()  
  
        pred = model(seq, seq_lens)  
        loss = bce_loss(pred, label)  
        loss.backward()  
        clip_grad_norm_(model.parameters(), max_norm=gclip)  
        optim.step()  
  
        num_corrects += ((pred > 0.5) == label).sum().item()  
        num_labels += label.numel()  
        total_loss += loss.item() * label.numel()  
  
    if i > 0 and i % 100 == 0:  
        print('Step {} / {}, Loss {}'.format(i, len(train_loader), loss.item()))  
  
    print('Training Loss: {}, Training Accuracy: {}'.format(  
        total_loss / num_labels, num_corrects / num_labels  
    ))  
    print('\nStarting evaluation...')  
    model.eval()  
    print('Evaluation Accuracy:', compute_accuracy(model, val_loader))  
    print('\n')
```

- As we have learned from the lecture, RNN may suffer from exploding gradients. To tackle this problem, we use a technique here called gradient clipping with the function `clip_grad_norm_`.



It rescales the gradients to keep them small and avoid taking a huge descent step.

Parameter tuning

Embedding dimension	Hidden size	gclip	epochs	Accuracy on test set
32	32	30	8	0.8426
50	40	30	8	0.8410
40	30	30	8	0.8508
43	33	30	8	0.8478
40	30	40	8	0.8430
40	30	40	5	0.8365

With this architecture, we achieved an accuracy of 85.1% with the parameters above.

Optional exercise: recurrent neural network

Implement LSTM

```
class LSTM(nn.Module):
    def __init__(self, input_size=1, hidden_size=20):
        super().__init__()

        #####

        self.hidden_size = hidden_size
        self.input_size = input_size

        #####
        # TODO:
        #####
        self.W_hh = nn.Linear(self.hidden_size, 4 * self.hidden_size, bias=True)
        self.W_xh = nn.Linear(self.input_size, 4 * self.hidden_size, bias=True)
        #####
        #                               END OF YOUR CODE                               #
        #####
```

In the initialization of LSTM, we need to define input weights and recurrent weights for the forget gates.

Implement LSTM

```
def forward(self, x, h=None, c=None):
    """
    Inputs:
    - x: Input tensor (seq_len, batch_size, input_size)
    - h: Hidden vector (nr_layers, batch_size, hidden_size)
    - c: Cell state vector (nr_layers, batch_size, hidden_size)

    Outputs:
    - h_seq: Hidden vector along sequence (seq_len, batch_size, hidden_size)
    - h: Final hidden vector of sequence(1, batch_size, hidden_size)
    - c: Final cell state vector of sequence(1, batch_size, hidden_size)
    """
    # Below code handles the batches with varying sequence lengths
    lengths = None
    if isinstance(x, PackedSequence):
        x, lengths = pad_packed_sequence(x)

    # State initialization provided to you
    state_size = (1, x.size(1), self.hidden_size)
    if h is None:
        h = torch.zeros(state_size, device=x.device, dtype=x.dtype)
    if c is None:
        c = torch.zeros(state_size, device=x.device, dtype=x.dtype)
    assert state_size == h.shape == c.shape

    # Fill the following lists and convert them to tensors
    h_seq = []
    c_seq = []

    #####
    # TODO: Perform the forward pass
    #####

    for xt in x.unbind(0):
        # Get the gates and update
        hiddens = self.W_hh(h) + self.W_xh(xt)
        gates, update = hiddens.split(
            (3 * self.hidden_size, self.hidden_size), dim=-1
        )
        gates = gates.sigmoid()
        update = update.tanh()

        # Update the hidden state
        fg, ig, og = gates.chunk(3, dim=-1)
        c = c * fg + update * ig
        h = og * c.tanh()

        # Keep the hidden state history
        h_seq.append(h)
        c_seq.append(c)

    h_seq = torch.cat(h_seq, 0)
    c_seq = torch.cat(c_seq, 0)

    #####
    # END OF YOUR CODE
    #####
```

We loop over the time step of the sequence and update parameters of the LSTM.

Questions? Piazza 😊